Counterexamples from Proof Failures in SPARK

David Hauzar, Claude Marché, Yannick Moy

▶ **To cite this version:**

David Hauzar, Claude Marché, Yannick Moy. Counterexamples from Proof Failures in SPARK. Software Engineering and Formal Methods, Jul 2016, Vienna, Austria. hal-01314885

HAL Id: hal-01314885
https://inria.hal.science/hal-01314885

Submitted on 12 May 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Counterexamples from Proof Failures in SPARK*

David Hauzar[1,2,3], Claude Marché[1,2], and Yannick Moy[3]

[1] Inria, Université Paris-Saclay, F-91893 Palaiseau
[2] LRI, CNRS & Univ. Paris-Sud, F-91405 Orsay
[3] AdaCore, F-75009 Paris

Abstract. A major issue in the activity of deductive program verification is the understanding of the reason why a proof fails. To help the user understand the problem and decide what needs to be fixed in the code or the specification, it is essential to provide means to investigate such a failure. We present our approach for the design and the implementation of *counterexample generation* within the SPARK 2014 environment, exhibiting values for the variables of the program where a given part of the specification fails to be validated. To produce a counterexample, we exploit the ability of SMT solvers to propose, when a proof of a formula is not found, a *counter-model*. Turning such a counter-model into a counterexample for the initial program is not trivial because of the many transformations leading from a given code and specification to a verification condition.

1 Introduction

Deductive program verification is an activity that aims at checking that a given program respects a given functional behavior. In this context, the expected behavior must be expressed formally by logical assertions, *i.e.* preconditions and postconditions, forming a *contract*. Deductive program verification typically proceeds by generating, from both the code and the formal specification, a set of logic formulas called *verification conditions* (VCs). If one proves that all generated VCs are tautologies, then the program is guaranteed to satisfy its specification. In recent program verification environments like Dafny [20], OpenJML [12] and Why3 [7], VCs are discharged using automated theorem provers, in particular those of the *Satisfiability Modulo Theories* (SMT) family such as Alt-Ergo [5], CVC4 [2] and Z3 [22]. These theorem provers are used as black-boxes that, given a VC, may produce three kinds of results:

1. The prover answers something meaning "yes, the VC is a tautology"
2. The prover answers anything else, meaning "I don't know", in order words the prover is not able to prove the VC for any reason
3. The prover runs for a too long time (seemingly infinitely) or runs out of memory

The case where the prover runs for too long time is handled in practice by setting a given time limit, so that the prover process is killed when exceeding this limit. The cases 2 and 3 are the same from the user's perspective: the VC is not proved. Note that

* Work partly supported by the Joint Laboratory ProofInUse (ANR-13-LAB3-0007, http://www.spark-2014.org/proofinuse) of the French national research organization

we do not distinguish a case where the prover would answer "no it is not a tautology", because the VCs typically involve undecidable logic features (*e.g.* non-linear integer arithmetic, first-order quantification) so provers are in practice incomplete: there is no way for them to be sure that a given VC is not provable.

A major issue in the activity of deductive verification is thus understanding the reasons for a proof failure. There are various reasons why it may fail:

1. The property to prove is indeed invalid: the code is not correct with respect to the given specification.
2. The property is in fact valid, but is not proved, again for two possible reasons:
   - The prover is not able to obtain a proof (in the given time and memory limits): this is the incompleteness of the proof search;
   - The proof may need extra intermediate annotations, such as loop invariants, or more complete contracts of the subprograms

For the user to be able to fix the code or the specification of their program, it is essential to understand into which of the two above cases any undischarged VC falls. The solution we propose in this paper is to generate *counterexamples*, or more precisely *potential* counterexamples. Such a counterexample should give values for the variables of the program, demonstrating a particular case where a given annotation may not hold. To produce a counterexample, we exploit an additional feature of SMT solvers: the ability to propose, when a proof of a formula is not found, a *counter-model*, exhibiting an interpretation of the free variables where the formula cannot be proved true. Turning such a counter-model into a counterexample for the initial program is not a trivial task because of the many transformations that lead to a VC from a given code and specification. For this work, our goal was to design and implement counterexample generation within the SPARK 2014 [21] environment for the development of safety-critical Ada programs. In this context, the initial program with annotations is first translated into the intermediate language WhyML. The Why3 tool [7] processes WhyML to generate verification conditions using a weakest precondition calculus. These VCs are then passed to SMT solvers after several possible transformations: simplifications and encoding of features not natively supported by SMT-LIB. Then, to turn the counter-model into a counterexample, one has to relate the model produced by the SMT solver back to the original problem, taking into account the entire transformation chain.

In Section 2 we present the support for counterexamples in SPARK 2014, from a user's point of view, illustrated by simple examples. In Section 3 we go into the internals of the tools, and explain how we designed our approach to generate counterexamples. We discuss related work and future work in Section 4. More details can be found in a technical report [16].

2 Counterexamples in SPARK

Ada 2012 is the latest version of the Ada language [1], a general purpose language, traditionally used in embedded software development. This version adds new features for specifying the behavior of programs, such as subprogram contracts and type invariants. SPARK is a subset of Ada targeted at formal verification [21]. Its restrictions

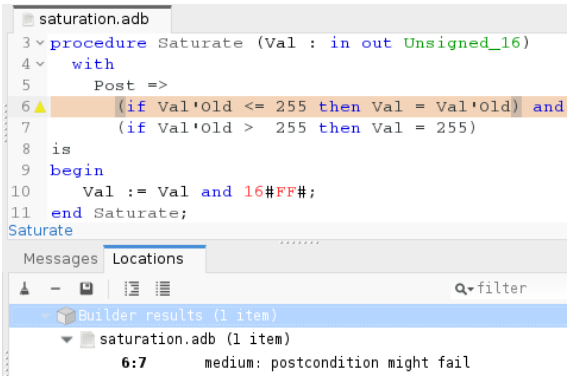

Fig. 1. A failed postcondition.

ensure that the behavior of a SPARK program is unambiguously defined. The SPARK language and toolset for static verification has been applied for many years in on-board aircraft systems, control systems, cryptographic systems, and rail systems [8]. SPARK also provides dedicated features that are not part of Ada 2012: essential constructs for deductive verification (*e.g.* loop invariants, ghost code) have been added. To formally prove a SPARK 2014 program, GNATprove uses WhyML as an intermediate language. The SPARK program is translated into an equivalent WhyML program which can then be verified using the Why3 tool.

Figure 1 shows an example of a saturation procedure, ensuring that values stay in a given range. In this example, the procedure should ensure that the output value is less or equal to 255. More precisely, the postcondition requires that if the input value is in the range, it is unmodified, and set to 255 otherwise. Note the attribute `'Old` that refers to the values that expressions had at procedure entry. The procedure is implemented using bit-wise AND with mask `0xFF`. As the message at the bottom shows, GNATprove does not succeed in proving the postcondition.

The means for the user to investigate the possible reason of the failure are:

- Execute code and properties during tests, in a way that violations of the property will stop execution with an exception. This depends of course on the availability of tests that exercise the violation, but testing is a well-known software engineering discipline that engineers usually master, hence uncovering incorrect code and properties is comparatively easier than investigating other reasons for proof failure.
- A focused manual review of the code and assertions can efficiently diagnose many cases of missing annotations.
- The user can try to increase the proof power along different axes, in order to combine the results of different provers and allocate more resources (in particular time) for each proof attempt. In GNATprove, in addition to the lower level switches, there are predefined *proof levels* between 0 and 4 that the user can increase to augment the proof power: more time allocated, use more provers.

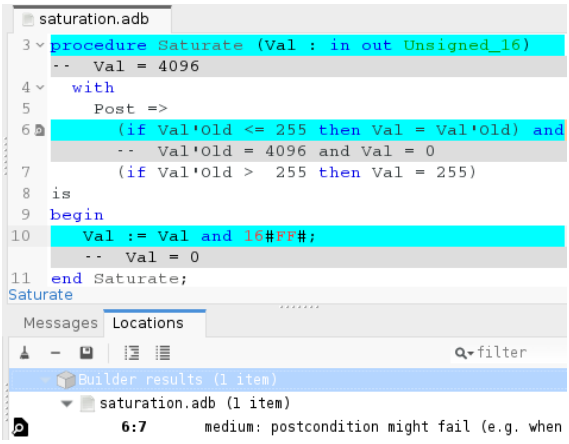

Fig. 2. Counterexample interleaved with code.

GNATprove also helps users by pinpointing the part of a larger assertion which is not proved, and the execution path along which the proof fails. During interactions, the IDE integration is of utmost importance to allow focusing the proof on a single subprogram or even a single line of code. Yet, testing and manual review may not identify all errors and missing annotations, and increasing the proof power may prove the property either. The burden is then on the user to verify the unproved property by other means: more tests, manual reviews, or using an interactive prover whose proof script is checked by GNATprove.

Adding Counterexamples in SPARK. We describe now the new facilities to generate counterexamples that is the purpose of this paper. There are multiple ways to integrate counterexamples in a development environment, depending on the expected degree of interactions with users. In SPARK, we have chosen to simplify the interactions to a minimum, so that users are directly presented with the most relevant information. GNATprove displays the values of relevant variables in the message displayed to the user for an unproved check. The message displayed by GNATprove on the example from Figure 1 is:

```
medium: postcondition might fail (e.g. when Val'Old = 4096 and Val = 0)
```

This information alone might be sufficient to understand the problem. Otherwise, GNATprove has pre-computed for every unproved check a counterexample trace that can be displayed in the IDE. This trace consists of a sequence of program lines, annotated with values of relevant variables.

For example, Figure 2 shows the trace computed by GNATprove and displayed in GNAT Programming Studio on the example seen before. A variable is selected as *relevant* in the summary message if it appears in the expression being checked. A variable on any given line is selected as *relevant* in the trace if it is assigned a new value on this line. As visible from Figure 2, the counterexample trace is displayed inside special

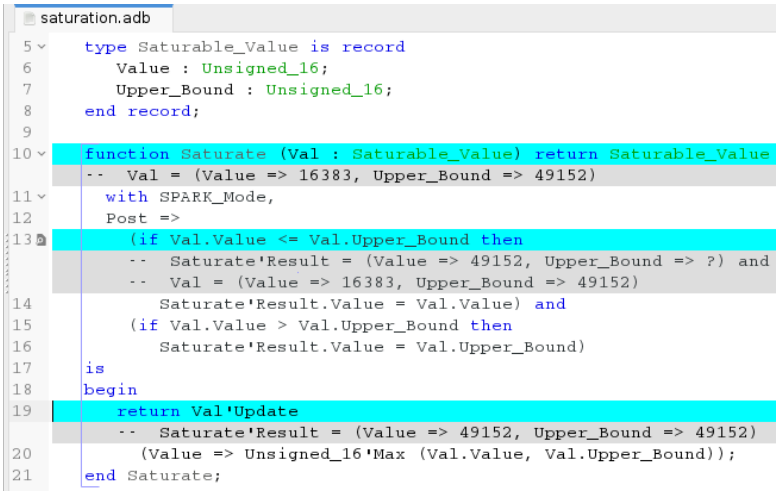

Fig. 3. Counterexample with a record type.

lines in the editor, that are not part of the code and cannot be edited manually (note the absence of a line number). These lines are prefixed with the token `--` that introduces comments in Ada code to make it clear to users that they are not part of the code. The lines in the program to which the trace applies (lines 3, 6 and 10) are emphasized in the editor. The counterexample shows that the implementation is indeed not correct with respect to the specification. Bitwise AND of `4096` and `0xFF` is `0`, while the specification requires that the returned value of `Val` be `255`.

Counterexamples with Records and Arrays. Counterexamples can contain values of record types and array types. Their values are displayed in the usual Ada syntax as aggregates, as illustrated in Figure 3. If the counterexample value of a field is not known, it is displayed as question mark. If there is more than one such field, then these fields are aggregated under the name `others`. On Figure 3, type `Saturable_Value` defined at line 5–8 contains a field `Value` representing the actual value and a field `Upper_Bound` being an upper bound of the saturation range. The postcondition of the function `Saturate` is analogous to the postcondition of the procedure `Saturate` from Figure 2. The field `Value` of the returned record must contain the value of the field `Value` of the input record if it is in the range, otherwise it must contain the upper bound of the range. The saturation is now implemented using function `Unsigned_16'Max`. The counterexample shows that if `Val.Value` is `16383` and `Val.Upper_Bound` is `49152`, `Saturate'Result.Val` is `49152`. Indeed, instead of the function `Unsigned_16'Max`, the function `Unsigned_16'Min` should be used.

Similarly for records, the content of arrays is shown in Ada syntax for array aggregates. For arrays with statically unknown ranges, the array range is also part of the counterexample, shown again in Ada syntax using the attributes `'First` and `'Last`. See the report [16] for a more detailed example.

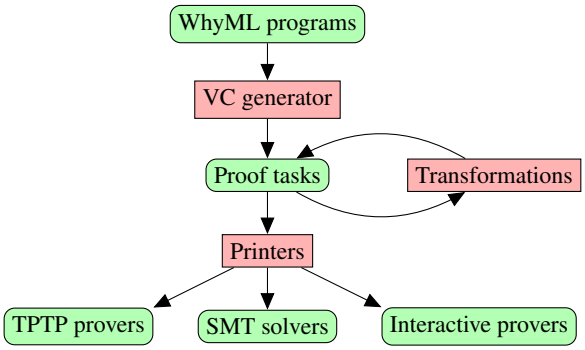

Fig. 4. Why3 architecture

3 Implementation of Counterexamples

3.1 Short Introduction to Why3

Why3 (`http://why3.lri.fr`) is an environment for deductive program verification, providing the language WhyML for specification and programming [14]. WhyML is used as an intermediate language for verification of SPARK programs as well as C and Java programs [13], and can also be used as a primary programming language (it can be compiled to Ocaml).

A schematic view of Why3's components is shown in Figure 4. The specification component of WhyML [6], used to write program annotations and background theories, is an extension of first-order logic. It features ML-style polymorphic types (prenex polymorphism), algebraic datatypes (in particular records), abstract types, and functions and predicates specified axiomatically. Why3 comes with a rich standard library providing general-purpose theories useful for specifying programs [7]. This includes integer and real arithmetic, arrays, and bit-vectors. The specification part of the language serves as a common format for theorem proving problems, *proof tasks* in Why3's jargon. The programming part of WhyML is a dialect of ML with a number of restrictions to make verification easier [14]. WhyML function definitions are annotated with preconditions and postconditions both for normal and exceptional termination, and loops are also annotated with invariants. Why3 generates proof tasks from user lemmas and annotated programs (using a weakest precondition calculus), then dispatches them to multiple provers. We detail below a few features of Why3 that are of particular interest for the counterexamples feature.

Transformations. A Why3 transformation is any procedure taking a proof task as an argument and producing another proof task, or more generally a set of proof tasks. Transformations must be *sound* in the sense that validity of the resulting tasks must imply the validity of the input task. The converse is generally true but not always. A typical example is the *split* transformation: for a given proof task of the form $H_1, \ldots, H_k \vdash \forall x.H \rightarrow (G_1 \wedge \cdots \wedge G_n)$, that is, if the goal ends with a conjunction, it produces the set of n tasks $H_1, \ldots, H_k \vdash \forall x.H \rightarrow G_i$ for $1 \leq i \leq n$. As most of the provers do

not support some of the language features, (*e.g.* pattern matching, polymorphic types, recursion), Why3 applies a series of encoding transformations to eliminate unsupported constructions before dispatching a proof task to provers. Other transformations can also be imposed by the user in order to simplify the proof search: inlining of definitions, simplification by computation, case analysis, application of inductive schemes, etc.

Labels. Why3 *labels* are arbitrary character strings, written between double quotes. They can be attached to any logic formula or term, and also to any declaration. Their interpretation is not fixed *a priori*; in some cases they are interpreted by specific transformations. For example, the *asymmetric conjunction* of Why3's logic is a connective written as `&&`. Internally, it is in fact the usual conjunction $\wedge$ with the label `"asym_split"` on the first argument. The `split` transformation interprets this label so that a goal of the form `f1 && f2` is split into the goals `f1` and `f1` $\rightarrow$ `f2`. Transformations that do not interpret labels keep them attached to formulas and terms, if possible. For example, a transformation may rename a variable, in that case it should propagate labels from the original variable to the new one. Analogously, if a transformation rewrites a given sub-term into another, it should also propagate labels of the old term to the new one.

Locations. To help traceability of errors from its various front-ends, WhyML has a mechanism of source locations similar to the `#line` directive of C pre-processor. Instead of being line-oriented, it is character-precise: any term or declaration can be given an annotation of the form `#`*`file l b e`*`#` meaning that this term or declaration originates from the source file *file*, at line *l*, from first character *b* to last character *e*. Similarly as for labels, transformations should propagate locations.

The Weakest Precondition calculus. The VC generator, which implements a variant of the weakest precondition calculus (WP for short), takes any WhyML function and creates a proof task. If that proof task is a tautology then the input function satisfies its contract. This formula is typically quite large, as it collects all the necessary checks that need to hold for the function to be safe: postcondition, but also initialization and preservation of loop invariants if any, any kind of runtime checks, etc. To present the resulting formula to the user in a more friendly manner, a default application of the `split` transformation is applied, so as to obtain a set of VCs that corresponds to the various checks to perform on the original program. To make this more user-friendly, Why3's WP calculus is instrumented so that each of the sub-formulas that corresponds to a program check is annotated with a label of the form `"expl:`*text*`"`. The *text* is an explanation of the VC, and is interpreted by the graphical interface. Regarding the counterexample feature, an important aspect is that during the computation of the WP, for each program statement that updates a program variable as a side-effect, a fresh logical variable holding this new value is created. This is the case for assignment statements, but also occurs in case of function calls and in presence of loops.

Metas. Why3's *metas* provide a way to associate metadata to a proof task that, unlike labels, are not attached to any particular sub-term or declaration, but are declared globally to the task. A meta is characterized by a name and a set of parameters that can be nearly of any kind of object: a number, a boolean, a string, but also a reference to

another declaration: a type, a function symbol, an hypothesis. As for labels, metas can be interpreted by transformations, but are usually kept unchanged. Unlike labels, the name of metas, and the type of their arguments, must be declared first.

3.2 Model Features of SMT-LIB

An SMT solver takes as input a set of formulas, and checks whether this set is satisfiable or not. To prove that a given proof task $H \vdash G$ is a tautology, we query the solver for the satisfiability of H and the negation of G: if the solver answers that this set is unsatisfiable, it means that proof task is valid. If the solver terminates with any other answer, the SMT solver may propose a potential model of H and $\neg G$ describing why $H \vdash G$ cannot be proved. To get such a model, we use features of SMT-LIB [3], and the solvers CVC4 and Z3. SMT-LIB defines commands `get-model` and `get-value` for getting models. The command `get-model` returns a set of interpretations for all user-declared function symbols in the input task. The command $(\texttt{get-value}\, t_1 \cdots t_n)$ returns for each term t_i a value term that is equivalent to t_i in the potential model.

3.3 Counterexamples at Why3 Level

Our goal is to exploit the generation of models by SMT solvers to construct a potential counterexample to the input Why3 program. This means that we need to add counterexample generation to the Why3 architecture described in Figure 4: some feedback from the bottom (prover results) to the top (input program) must be implemented. Because the VC generation and the Why3 transformations can rename variables and introduce fresh ones, re-interpreting the model returned by the solver into a counterexample of the input source is a non-trivial process.

A first choice we have to make is on whether using the `get-model` or the `get-value` command of SMT-LIB. The command `get-model` might seem easier to use at first because no argument needs to be given. However, from the large set of function symbols and their values returned by `get-model`, it would be a hard task to extract which part of it corresponds to the initial program, because we have no trace of the extra logical variables and renamings made by WP and transformations. That's why we decided to use the `get-value` command instead. We provide the variables or terms to query as arguments of this command by properly propagating traceability information along the WP and the Why3 transformations. This is done using Why3 labels and metas instrumenting the different processing steps as shown in Figure 5. This has to be performed regarding different aspects that are detailed in the subsections below.

Marking variables to show in a counterexample. In a Why3 task, variables that should be shown in a counterexample are marked with the label `"model"`. When the task is printed into SMT-LIB format, SMT-LIB terms corresponding to these variables are collected and then passed as parameters of the `get_value` command. As an example, see the following Why3 task:

```
constant x "model" :int
goal G : x+x > 0
```

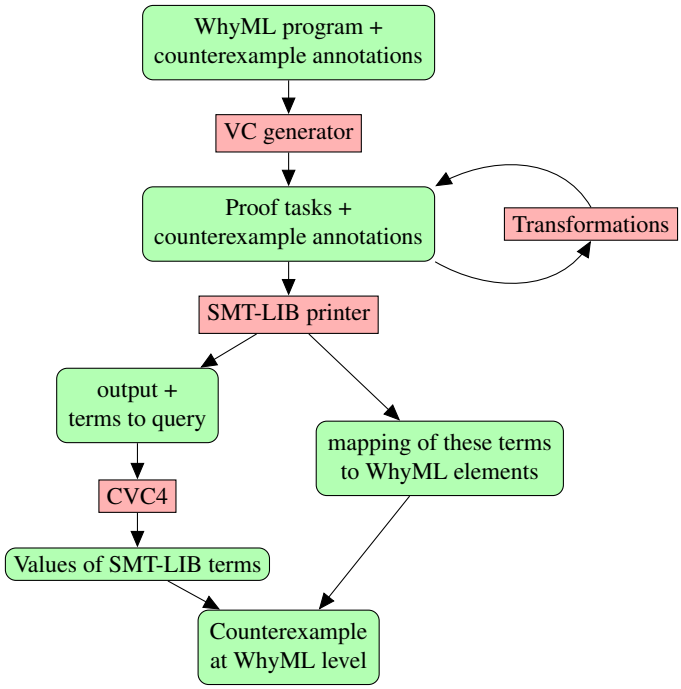

Fig. 5. Counterexamples at Why3 level.

When printing the task into SMT-LIB syntax, the SMT-LIB term corresponding to the constant `x` will be collected and queried for counterexample value `v`. The counterexample will be displayed to the user in the form `x = v` and this equality will be associated to the location of the goal `G`. For a Why3 task that is generated from WhyML or SPARK program, we additionally need to annotate each variable with two things. First, with a location in the original source code and second with the name of the variable in the source language.

```
constant x "model" "model_trace:X" #file.adb 42 1 2#:int
goal G : x+x > 0
```

In such a case the counterexample will be displayed in the form `X = v` and associated with location in file `file.adb`, line `42` (in practice inside a comment as in Figure 2).

Instrumenting WP calculus for counterexamples. The user expects that all successive values of a variable, marked with label `"model"`, appear in a counterexample. WP creates a fresh logical variable for every modification of a given variable. For variables marked with label `"model"`, counterexample labels are propagated to these fresh logical variables. Moreover, each of these fresh variables is given the location of the expression that triggers its creation. As an example, let us consider the following WhyML code implementing a simple loop:

```
let test_loop ( x "model" "model_trace:X" : ref int ): unit
  requires { !x > 0 }
= while !x > 0 do                          (* counterexample: X = 1 *)
    invariant { !x ≥ 0 }
    x := !x - 2                           (* counterexample: X = -1 *)
  done
```

The variable x is marked with labels "model" and "model_trace" as counterexample variable. The property preserving loop invariant is not proved, and a counterexample, shown in comments, is generated. The formula encoding this property is shown in Figure 6. The variable x quantified at the top of the formula stands for the input value of the variable x of the test_loop function. Then, WP creates another fresh variable x1 for the value of variable x at the beginning of some arbitrary loop iteration. Finally, WP creates a fresh variable x2 for the value of variable x after the assignment statement.

As shown on Figure 6, the "model" label on the variable x of test_loop is propagated to all those logical fresh variables corresponding to x. A similar propagation occurs with label "model_trace" with some additional information (after the "@" sign) to explain the origin of the fresh variable. Source code locations are not displayed here for readability.

```
forall x "model" "model_trace:X".
 (x ≥ 2) →
  (forall x1 "model" "model_trace:X@loop".
   x1 ≥ 0 →
    x1 > 0 →
     (forall x2 "model" "model_trace:X@call".
      (x2 = (x1 - 2)) → x2 ≥ 0)))
```

Fig. 6. Logical formula generated by WP.

Get values of variables from a given assertion. In practice, it is useful for the user to see values of counterexample variables at the location of the assertion that fails. As an example, see Figure 2. Both initial and final value of variable Val are displayed on line 6, which is the location of the failed postcondition and this information is also a part of the message summarizing the unproved assertion in the "messages" panel.

```
let test_post
    (x "model" "model_trace:X" : int)
    (y "model" "model_trace:Y" :
                      ref int): unit
          (* counterexample: X = 1 *)
 ensures { "model_vc" !y ≥ x }
   (* counterexample: Y = 0, X = 1 *)
= y := x - 1
          (* counterexample: Y = 0 *)
```

Fig. 7. Counterexample located at postcondition annotated with "model_vc"

During WP, all modifications of variables relevant for counterexamples are marked, and their values are displayed at the respective locations that triggered the modifications. However, the user expects to see also, at the location of a failed check, the values of the variables involved in that check. One way to display these variables at the location of a failed check would be to retrieve the last point of modification of a variable and display this counterexam-

ple value at the location of a failed proof. However, this is quite complex to do when multiple program paths are encoded in a VC.

That is why we preferred to explicitly mark variables that appear at the location of a failed check. In WhyML programs, expressions that trigger generating a proof task are marked with label `"model_vc"`. These expressions can be asserts, preconditions, and postconditions. We implemented a dedicated Why3 transformation that uses this label to find the expression that triggers generating the current proof task. The transformation then marks all counterexample variables read in this expression as a part of the counterexample at locations of the expression.

The example of Figure 7 shows a function with a postcondition marked with a label `"model_vc"`. This postcondition cannot be proved and a counterexample is generated. At the location of the postcondition, the values of the variable `x` at the function start and the variable `y` at exit are displayed.

Projections in models. For some types, SMT-LIB standard does not specify how values of these types should be displayed. Most notably, this is the case for abstract types. When an SMT solver is queried for values of such a type, it usually returns just an internal reference. To display values of these types in a counterexample, we decided to *project* them to values of types that can be displayed. To project values of a type T_1 to a type T_2, a meta `"model_projection"` must be declared taking as argument some function P_1 from T_1 to T_2. If some element E of type T_1 is labeled with `"model_projected"`, then instead of querying for a value for E, the solver is queried for a value of $P_1(E)$. Projections are applied transitively: if there is a projection function P_2 from T_2 to T_3, a value of $P_2(P_1(E))$ is queried. Moreover, if there are more than one projection for a projected type, all of them are applied. Projecting values is implemented as a Why3 transformation `intro_projections_cntexmp`.

```
type byte
function to_rep byte : int
predicate in_range (x : int) =
  -128 ≤ x ≤ 127
axiom range_axiom : forall x:byte.
  in_range (to_rep x)
meta "model_projection"
  function to_rep
constant a "model_projected"
           "model_trace:A" : byte

type r = {f : byte; g : bool}
function proj_f
  "model_trace:.F" (x:r) : byte =
     x.f
meta "model_projection"
  function proj_f
function proj_g
  "model_trace:.G" (x:r) : bool =
     x.g
meta "model_projection"
  function proj_g
constant b "model_projected"
         "model_trace:B" : r
```

Fig. 8. Projections of abstract types and records.

Figure 8 defines an abstract type `byte` to represent integers from -128 to 127. Values of this type can be projected to integers using function `to_rep`. This function is marked as a projection using meta `"model_projection"`. The variable `a` of type `byte` is marked with the label `"model_projected"`. This means that `a` will be queried in counterexamples and will be projected from `byte` to `int` using `to_rep`.

Querying record values reuses projection mechanism to extract their fields. For each field, a projection function is defined, marked using meta `"model_projection"`, and annotated with a `"model_trace"` label specifying the name of the field. When the transformation `intro_projections_cntexmp` uses this function to project a record value to the record field, it adds the name of the field to the content of `"model_trace"` label of the record value. Remember that projections are applied transitively: if a field is of a type with a defined projection, it is further projected. Figure 8 shows an example of definition of record type `r` with fields `f` and `g`. Functions `proj_f` and `proj_g` project a value of type `r` to field `f` and `g` and they are annotated with `"model_trace"` labels capturing the names of the fields that will be displayed in a counterexample. The constant `b` is marked to be queried for a counterexample with `"model_projected"` label meaning that the value must be projected before being displayed and it is annotated with `"model_trace"` label that captures the name of the variable that will be displayed in a counterexample.

Arrays. SMT-LIB does not define how values of array types should be output in a counterexample. To get values of array types, we rely on the form in which values of array types are returned by the CVC4 solver: an array as a constant array and series of store operations defining relevant indices. Here are two examples of array values that CVC4 may return:

```
(store (store ((as const (Array Int Int)) 0) 1 2) 3 4)
((as const (Array Int (Array Int Int))) ((as const (Array Int Int)) 0))
```

The first array is a single-dimensional array with index 1 equal to 2, index 3 equal to 4, and other indices equal to 0. The second array is a two-dimensional array with all indices equal to 0. The values stored in the array may be of abstract or record types so we need to project them. The problem is that we cannot proceed as for records by introducing projections for each array index because there are infinitely many of them. To overcome this problem, for an array `orig_arr` that should be queried for a counterexample and has values of an abstract type `t_val`, a projection function `pf_val` from the abstract type `t_val` to concrete type `t_val_c` is defined. Then, new array `proj_arr` with values of the type `t_val_c` is defined together with an axiom stating that projections of values in the original array are equal to the values in the new array:

```
constant proj_arr: map int t_val_c
axiom proj_axiom : (forall i : int. proj_arr[i] = pf_val(orig_arr[i]))
```

Instead of querying the solver for the original array, the solver is queried for the new, projected array.

3.4 Building Counterexamples for SPARK

A SPARK program is translated by the tool `gnat2why` into a WhyML program with counterexample annotations. Why3 generates VCs and tries to prove each resulting proof task with selected provers. If all fail, the task is split into smaller tasks. When a task can be neither proved nor split, it is attempted to be proved in the counterexample mode described in Section 3.3. The generated counterexample is returned back to `gnat2why` and post-processed, before being displayed to the user.

Generating WhyML code. `gnat2why` marks all WhyML elements corresponding to declarations of SPARK variables or to declarations of arguments of SPARK functions to be part of a counterexample using `"model"` or `"model_projected"` labels, generates projection functions for abstract and record types generated by `gnat2why` and marks WhyML elements that trigger generating of a VC by `"model_vc"` labels. `gnat2why` also generates `"model_trace"` labels storing traceability information to corresponding elements in SPARK program. Instead of storing names, `"model_trace"` labels store unique identifiers from SPARK internal representation (AST). `gnat2why` generates Why3 location tags, which make it possible to explicitly specify source code locations of WhyML elements.

Post-processing counterexamples. The counterexample returned from Why3 to `gnat2why` is a map from locations in SPARK source code to lists of counterexample elements at these locations. A counterexample element consists of an identifier and a value. The identifier has the form $x.f_1 \dots f_n$ ($n \geq 0$) where x (resp. f_i) is the internal AST identifier of a variable (resp. field). Counterexample elements are post-processed in the following way: identifiers are mapped back to names in the source code, elements in the same source code line corresponding to same record are grouped together as an aggregate (as in Figure 3). Values are converted to SPARK syntax.

3.5 Experimental Evaluation

Our implementation of counterexample generation is publicly available in Why3 0.87 and SPARK 16.0. On the full SPARK regression test-suite consisting in 1472 tests, enabling counterexample generation only induce a small slowdown if any on all supported platforms.

Figure 9 presents the results on the section of the test-suite that was initially created for the Riposte tool [25], which was used in the previous versions of SPARK to generate counterexamples. Overall, in most of the cases, counterexamples were obtained and they were of a good quality. The main difficulty for counterexample generation was the use of non-linear arithmetic (tests `arithmetic`, `alpha_launch_examples`, and `victor_divmod_rules`) and the presence of arrays (tests `array_aggregates`, `arrays`, `simple_arrays`, `arrays_multidim`, `array_application`, and `complex_arrays`). In the case of arrays, this is likely caused by the additional projections and axioms that are generated for arrays when generating counterexamples, as described in Section 3.3.

4 Conclusions and Perspectives

We added the generation of counterexamples to SPARK 2014, by exploiting the model generation feature of SMT solvers, and appropriately instrumenting the process of generating VCs from a SPARK program, through the intermediate WhyML program, weakest precondition calculus and logic transformations. Instead of complex post-processing of the complete model that would be returned by the SMT-LIB `get-model` command, we instrumented the processing steps so that only the adequate terms are queried with

Test	VCs	Unproved	Counterexamples		Good Counterexamples	
			Number	Percentage	Number	Percentage
basic	42	4	4	100%	4	100%
logic	46	11	11	100%	11	100%
enums	34	4	4	100%	4	100%
real_world	10	4	4	100%	4	100%
mixed	9	2	2	100%	2	100%
array_algorithms	44	2	2	100%	2	100%
records	115	19	18	95%	18	95%
alpha_launch_examples	17	8	6	75%	6	75%
array_aggregates	172	25	19	76%	19	76%
arrays	51	13	12	92%	8	62%
simple_arrays	109	50	50	100%	30	60%
usergroup_examples	15	4	2	50%	2	50%
victor_divmod_rules	58	9	4	44%	4	44%
arithmetic	126	24	10	42%	10	42%
arrays_multidim	20	13	10	77%	2	15%
array_applications	44	9	1	11%	1	11%
complex_arrays	39	10	10	100%	0	0%
All	951	211	169	80%	127	60%

Fig. 9. Results of counterexample generation on Riposte tests.

the `get-value` command, and then a simple mapping from the terms queried to the initial program variables can be applied to build the counterexample.

Recent user training sessions showed a clear appeal of counterexamples to users, which motivated our choice to enable them by default in SPARK (versions 16.0 and later). Based on our initial feedback with the use of counterexamples inside SPARK, counterexamples may be the most useful feature in SPARK for investigating unproved properties, after the possibility to execute contracts and assertions in tests.

Related Work. The model returned by a SAT or SMT solver on a satisfiable problem is exploited in several areas of program verification, a major case being the one of model checking, as for example in the Alloy analyzer [26] or the CBMC model checker for C programs [15]. In the case of deductive verification, generating counterexamples is not as common. The Riposte tool based on answer set programming [25] was used in the previous versions of SPARK to generate counterexamples, but only at the level of VCs without source traceability. There is also the case of the NitPick tool inside the Isabelle proof assistant [4].

In the more specific case of program verifiers using SMT solvers, the idea of instrumenting the generation of VCs originates from the old system ESC/Modula-3, that generates VCs for the Simplify solver, adding specific labels to determine the source location and the path of execution leading to the potential program error. The same mechanism was reused in ESC/Java [19]. The potential counterexample proposed by Simplify can be displayed to the user, but is very hard to understand because of the

various encodings from the input program to the VC. Only recently a way to reinterpret the counterexample in terms of variables of the source code was designed in the OpenJML framework [12]. They use SMT-LIB command `get-value` to get counterexample values for all sub-expressions in the original program, supporting values of scalar types only, and also to get values of block predicates, which they use to determine the control-flow path of the failed assertion [11]. In SPARK, it is possible to generate VCs for individual control-flow paths and display control-flow path for such VCs if they cannot be proved. In OpenJML, SMT-LIB VCs are generated directly, without using intermediate representation. On one hand, this make it easier to maintain mapping between source-code variables and logical variables. On the other hand, using Why3 as intermediate language makes it possible to use the power of Why3 transformations to transform a proof task to forms well suited for different provers. Another deductive program verification framework that makes use of SMT counter-models is the Boogie Verifier Debugger [18]. Boogie is used as an intermediate language by Dafny [20] and VCC [10]. Boogie also has its own way of reinterpreting the counter-model, generated by its back-end prover Z3, in terms of the source code. Besides scalar values, Boogie makes it possible to display the content of dynamically allocated data structures such as objects. Unlike SPARK and OpenJML, Boogie encodes locations and source variable names in the generated VC, uses SMT-LIB command `get-model` to get whole SMT-LIB counterexample and then relies on reverse transformations to map the SMT-LIB counterexample into the source code.

Both OpenJML and Boogie present the counterexample in a user-friendly manner, in their respective graphical interfaces (Eclipse, Visual Studio). Their presentation is a bit different from our way of presenting the counterexample, where we give values of relevant variables inside comments at proper locations of the source code. We have no evidence that our approach is better than these other approaches in terms of quality of the generated counterexamples. We designed our approach so that it is the best fit for SPARK users.

Another recent approach for helping users in debugging their specification and code is to use some kind of symbolic execution, as is proposed by the Visual Studio dynamic debugger [23] and the Verifast verifier [17].

Future Work. During this work, we encountered a few issues that could be addressed by authors of SMT solvers.

First, SMT-LIB standard does not fix any rule for displaying model values. In particular, it is not standardized how values of array types and bit-vector types should be displayed. This need for standardization is already known and it is likely to appear in the near future. Related to this, we believe that the feature of projections that we introduced could be handled by the solvers themselves as part of the standard to display counterexamples. This would be particularly useful in the case of arrays: the solution we proposed, involving the introduction of another array and an axiom, makes the problem harder to prove because of the additional universal quantification.

A second issue concerns the validity of generated counterexamples. In principle, one should query SMT solvers for models only if the answer was `'sat'`. However, on a VC generated by a program verification task, most of the time the answer is `'unknown'` or

the solver hits the time limit given. As expected, in this case the model is not guaranteed to be a true model. However, there are some cases where the model returned is *trivially wrong* because it is not even a model of the ground part of the goal. A suggestion for improvement is as follows: since the main source of incompleteness comes from the quantified hypotheses, there could be two different modes of operation, with two corresponding time limits. A first time limit, say a "soft" one, gives the time during which the solver is allowed to instantiate quantifiers as it wants. After this soft time limit is reached, a "hard" time limit should give the solver extra time to continue its search but in a specific mode where no new quantifier instantiation is performed. In this second mode, it is likely that the solver would terminate its search, and if a model is returned, it would be valid with respect to the ground part of the goal. If such modes were implemented in SMT solvers, it would be of major interest for counterexample generation.

Another technical issue is the ability to support model generation for all supported theories. This is not always the case, for example CVC4 does not produce models when non-linear arithmetic is selected. It is understandable since this logic is undecidable, there is no way to be sure that the model returned would be a true one. However, a similar degraded mode as described above could be implemented, for example in the degraded mode non-linear parts of the formulas could be ignored.

To double-check that a counterexample produced by our technique is a true one, one may consider turning it into a test case and run the program with the given values. This is unfortunately not an easy task because of the procedure calls: a procedure has a concrete semantics given by concrete execution and abstract semantics given by contracts. Since only the abstract semantics is visible to a solver, it may happen that a counterexample is true with respect to the abstract semantics, but false with respect to the concrete semantics and moreover it can happen that there is a different counterexample, not returned by the solver, true with respect to both semantics. Thus, properly combining counterexamples generated by failed proof attempts and run-time verification needs to investigated further. Recent work by Christakis et al. [9] and Petiot et al. [24] pursue such a direction.

Acknowledgements. We would like to thank David Cok, Clément Fumex, Rustan Leino, Andrei Paskevich, Florian Schanda, as well as the anonymous reviewers for their useful comments. We are pleased that a reviewer specifically agreed with us on "the suggested improvement to SMT solvers regarding hard and soft limits" and another confirmed that "the insights discussed as future work are very interesting".

References

1. Barnes, J.: Programming in Ada 2012. Cambridge University Press (2014)
2. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Computer Aided Verification. pp. 171–177. Springer (2011)
3. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: 8th Int. Workshop on Satisfiability Modulo Theories (2010)
4. Blanchette, J.C., Nipkow, T.: Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In: Interactive Theorem Proving. LNCS, vol. 6172, pp. 131–146. Springer (2010)

5. Bobot, F., Conchon, S., Contejean, E., Iguernelala, M., Lescuyer, S., Mebsout, A.: The Alt-Ergo automated theorem prover (2008), http://alt-ergo.lri.fr/
6. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Int. Workshop on Intermediate Verification Languages. pp. 53–64. Wrocław, Poland (2011)
7. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Let's verify this with Why3. International Journal on Software Tools for Technology Transfer 17(6), 709–727 (2015)
8. Chapman, R., Schanda, F.: Are we there yet? 20 years of industrial theorem proving with SPARK. In: Interactive Theorem Proving. LNCS, vol. 8558, pp. 17–26. Springer (2014)
9. Christakis, M., Leino, K.R.M., Müller, P., Wüstholz, V.: Integrated environment for diagnosing verification errors. In: Tools and Algorithms for the Construction and Analysis of Systems. Springer (2016)
10. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Theorem Proving in Higher Order Logics. LNCS, vol. 5674. Springer (2009)
11. Cok, D.R.: Improved usability and performance of SMT solvers for debugging specifications. Int. Journal on Software Tools for Technology Transfer 12(6), 467–481 (2010)
12. Cok, D.R.: OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse. In: Formal Integrated Development Environments. Elec. Proc. Theoretical Computer Science, vol. 149, pp. 79–92 (2014)
13. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Computer Aided Verification. LNCS, vol. 4590, pp. 173–177. Springer (2007)
14. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: European Symposium on Programming. LNCS, vol. 7792, pp. 125–128. Springer (2013)
15. Groce, A., Kroening, D., Lerda, F.: Understanding counterexamples with explain. In: Computer Aided Verification. LNCS, vol. 3114, pp. 453–456. Springer (2004)
16. Hauzar, D., Marché, C., Moy, Y.: Counterexamples from proof failures in the SPARK program verifier. Research Report 8854, Inria (2016), https://hal.inria.fr/hal-01271174
17. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: NASA Formal Methods. LNCS, vol. 6617, pp. 41–55. Springer (2011)
18. Le Goues, C., Leino, K.R.M., Moskal, M.: The Boogie Verification Debugger. In: Software Engineering and Formal Methods. LNCS, vol. 7041, pp. 407–414. Springer (2011)
19. Leino, K.R.M., Millstein, T., Saxe, J.B.: Generating error traces from verification-condition counterexamples. Science of Computer Programming 55(1–3), 209 – 226 (2005)
20. Leino, K.R.M., Wüstholz, V.: The Dafny integrated development environment. In: Formal Integrated Development Environments. Elec. Proc. Theoretical Computer Science, vol. 149, pp. 3–15 (2014)
21. McCormick, J.W., Chapin, P.C.: Building High Integrity Applications with SPARK. Cambridge University Press (2015)
22. de Moura, L., Bjørner, N.: Z3, an efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 4963, pp. 337–340. Springer (2008)
23. Müller, P., Ruskiewicz, J.N.: Using debuggers to understand failed verification attempts. In: Formal Methods. LNCS, vol. 6664, pp. 73–87. Springer (2011)
24. Petiot, G., Kosmatov, N., Botella, B., Giorgetti, A., Julliand, J.: Your proof fails? testing helps to find the reason (2015), http://arxiv.org/abs/1508.01691
25. Schanda, F., Brain, M.: Using Answer Set Programming in the Development of Verified Software. In: Technical Communications of the 28th Int. Conf. on Logic Programming. LIPIcs, vol. 17, pp. 72–85. Leibniz-Zentrum fuer Informatik (2012)
26. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: TACAS. pp. 632–647 (2007)